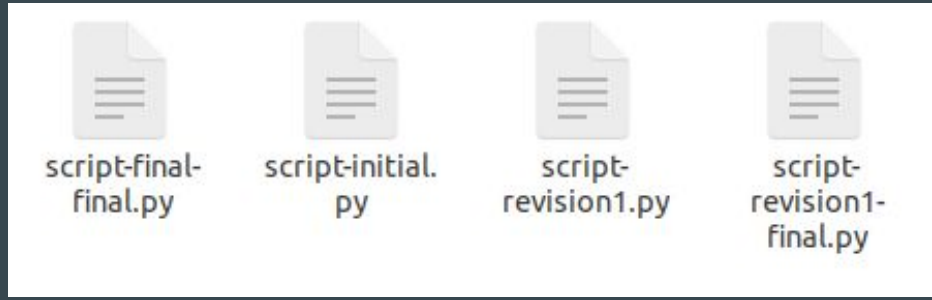
Version Control and Backups

Liam Porr

Why Version Control?



- Keep track of working versions
- Prevent other people from overwriting your changes
- Create and test features without breaking production

- Examples: Mercurial, Subversion, Git
  - Git is the most popular, so we'll focus only on it

Theory of Git

- Track changes in sets called commits
- Each commit is chained to the previous commit, and they form what we call a “tree”
- Each commit is part of an immutable tree that tracks a complete history of changes
- Can view the state of the project at any point in history by selecting any commit in the tree

Getting Started in Git

- We have a folder, `project/`, we want to use Git in
- `cd project`
- **`git init`**
  - This makes `project/` a Git “repository”

OR

- **`git clone [repo url] [destination folder]`**
  - Download a Git repository from elsewhere (Github)

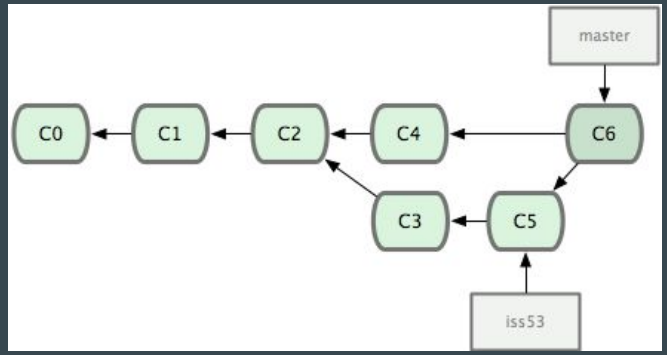
Adding Files and Committing

- By default, Git only takes care of files you tell it to--"tracked files"
- Imagine `project/file1`, `project/file2`
- Make some changes, then:
- **`git add file1 file2`**
  - Git now manages the history of these two files
- Commits
  - A set of changes to tracked files
  - The basic unit of history in Git
  - Think of it as saving state
- **`git commit -m "made some changes"`**
- **Commit early, commit often!**

Local Git Workflow

- To prepare a file for commit, you must stage it first
- **git stage my_file**
  - **(git add my_file works too)**
- This tells git that you want to include changes to this file in the commit
- Now commit with
  - **git commit -m “changed my_file”**
- Modify -> Stage -> Commit
- **Question** : are there any issues that could arise from this model?

Branching, Merging, Rebase



- Branching: literally make a branch from the ‘trunk’ of the tree
  - A series of new commits that shares a history with the mainline
  - **`git checkout -b <name of branch>`** (create a new branch and switch to it)
- Merging: combine a branch back into the mainline/trunk:
  - Creates a merge commit
  - on the head branch (trunk equivalent), do **`git merge <name of branch>`**
- Rebase: take all commits from a branch and apply them on top of the head
  - No merge commit
  - As if you just made all the commits on the main branch anyways
  - Often considered “cleaner” in terms of viewing git history
  - On the trunk branch, do **`git rebase <feature branch>`**

Pulling and Pushing

- The remote/offsite copy of the repository is called a “remote”
  - Github, Bitbucket
  - Someone else’s computer
- View remotes by doing **`git remote -v`**
- You can push or pull branches from remotes, and you can fetch changes (download them, but don’t integrate them into your tree)
- **Question** : How can I avoid having to copy the git history over every time I synchronize with a remote server?

Summary: The Git Workflow

In a git repository:

1. git checkout master
2. git checkout -b feature-branch
3. Edit some files
4. git stage files-ive-edited other-files-ive-edited
5. git commit -m “message about the edits I’ve made”
6. git push origin feature-branch
7. git checkout master
8. git merge feature-branch


BACKUPS
YOUR SHIT WILL BREAK

Backups

- Just do it
- Murphy's law
  - Accidental or malicious deletion
  - Device failure
  - Software failure
  - And a Berkeley special, theft.
- Automated backups because you will forget
- Don't leak information! Backups must be secure
- Make sure your backups actually work!
  - Routinely test backup procedure and recovering from backups

The 3-2-1 Rule

3. Have at least 3 copies of your data

2. Store your data on at least 2 different media
   >1 hard drive, >1 backup server/computer

1. Have at least 1 copy of your data off-site
   E.g. on Amazon S3, “the cloud,” under a mattress

What happens if you don't follow what I said

- GitLab 1/31/17 Database Outage
- Engineer accidentally runs `rm -rf` on their production PostgreSQL database
  - Noticed after 1 second and CTRL-C'd, but lost 300GB of production data already
- This shouldn't have been too bad - 300GB isn't THAT much data nowadays, and they can just recover from a backup, right?

Backup 1: Amazon S3

- GitLab had an automated process to upload a backup to Amazon S3 (Amazon file storage) every 24 hours
- GitLab engineers inspected their S3 bucket, hoping to find a backup
- Turns out their backups had been failing for weeks due to a version mismatch, and their notification system was broken too

Backup 2: Azure Disk Snapshots

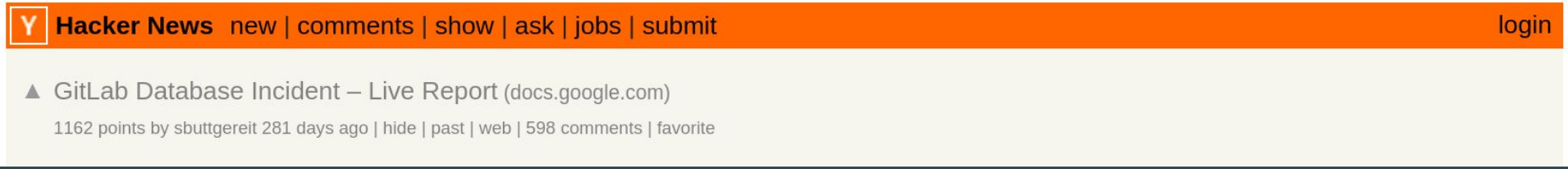
- GitLab runs on Azure (Microsoft Cloud Hosting Provider)
- Azure offers the option to generate snapshots of an entire disk periodically
- GitLab had enabled this to run every 24 hours...
- ....except on the database servers, because they thought they had enough backups

Why GitLab still exists today: Hail Mary LVM Snapshots

- LVM: Logical Volume Manager
- Not meant to be a backup, but luckily they had these
- Every 24 hours, copy data from prod to staging environment
- An engineer had run this ~6 hours before the incident luckily enough
- Unfortunately, took GitLab **18 hours** to recover, since staging was not meant for data recovery process
    - Different region and slow disks

Impact

- “It's hard to estimate how much data has been lost exactly, but we estimate we have lost at least 5000 projects, 5000 comments, and roughly 700 users.”
- Became both a feel-good story of transparency and not firing the engineer involved but also a WTF story about their backups on HackerNews
- Good lesson on the importance of keeping backups, making sure they work, and practicing recovering from them



Tools for Backups

- rsync
  - Simple command-line util for local <-> remote transfer
  - Skips copying files that are the same @ destination, so good for backups
  - Uses SSH for transferring to remote hosts
  - `rsync -av -P [source] user@host:[destination]`
- Rclone: “rsync for cloud storage”
  - Supports every major cloud storage provider
  - `rclone sync source:path dest:path`
  - Can mount cloud storage as local FS

More Tools

- Rsnapshot
  - Uses rsync, but effectively incremental backups that look full
  - Good for storing multiple backups (-1d, -3d, -1w, etc.) w/o using too much disk space
  - Other incremental-backup tools: Borg, Duplicity
- Some people think its a good idea to use git for backups ... dont.
  - Git is only meant for small files, text files really.
  - Using git for large files will take up lots of unnecessary disk space.

Conclusion

- Back up your shit